第5章

プログラミングにおける良いデータ構造とは

――新人技術者のためのロジカル・シンキング入門(2)



冴木 元

ここでは，組み込みソフトウェアにおけるデータ構造とその管理方法について解説する．どのようなデータ構造をとるかをシステム開発の早い段階で決めておくことで，保守性の高いプログラムを設計できる．指針を立てるためには，自分が設計するシステム，あるいは機能ブロックがどのような性質のデータを扱うのか，プログラミング言語で実装するにはどの種類の変数(外部変数，ローカル変数，テーブルなど)を使えばよいのかを十分に検討することがたいせつである．（編集部）

Bさんは，とあるソフトウェア・ハウスの中堅SE(システム・エンジニア)です．Bさんは，あるプロジェクトのヘルプに途中から入りました．そのとき，「関数単体で動作を確認するとOKなのに，結合するとうまくいかない機能ブロックが多い．ソース・レビューを近々やろうと思うので参加してほしい」と言われました．

● どこもかしこも外部変数

Bさんはプロジェクトに参加するにあたって，ソース・コードを資料として渡されていました．「どれどれ」と思いながらプログラムを読み進めたBさんは，まもなく頭を抱えることになります．外部変数と思われる変数があちこちに散らばっていて，関数の挙動が読みにくかったからです．

外部変数であるということは，ソース・コードのどこからでもリード/ライトが可能ということです．そのため，これらのソース・コードをデバッグしようとすると，ソース・コードの中にかたっぱしからgrep[注1]をかけて，ソース・コードのどこからのアクセスが最初かを特定しなければなりません．

立ち上げ時からこのプロジェクトを率いているベテランSEのPさんにこのことを話すと，「そのとおりなんだ．若い連中は開発の基本がわかっていないよね．もう一度基本に立ち返って開発を立て直すためにレビューが必要なんだ」とのことでした．

でもよく考えてみると，Pさんがそれなりに早い段階で方針を定めていれば，ソース・レビューを行う以前に，こういうソース・コードは生まれなかったのではないでしょうか…．「Pさんに評論家以上の役割を期待するのは無理かな」と，Bさんはふとため息を漏らしてしまうのでした．

これからBさんが担当する機能ブロックは，既存のコードを多数流用する必要があります．もし開発の切れ目で時間がある程度とれるのであれば，なんとか保守性を高めたソース記述にしたいのですが，どうすればよいのでしょうか？

1 外部変数の役割とは

外部変数の管理は，古くて新しい問題です．エピソード中のPさんのように，無秩序に外部変数を使用することが有害であると頭ではわかっていても，明確な方針を定めて保守性の高いプログラムをチームで開発できない，という人は少なくないように思います．外部変数が散乱するという事態は昔からありがちなのですが，その問題は意外と根が深いのです．

注1：特定文字列の照合(パターン・マッチング)を行うコマンド．

KeyWord
組み込みソフトウェア，データ構造，外部変数，テーブル，スタティック・データ，ローカル変数，ダイナミック・データ，スタック・オーバフロー，テスト・ケース，コーディング規約

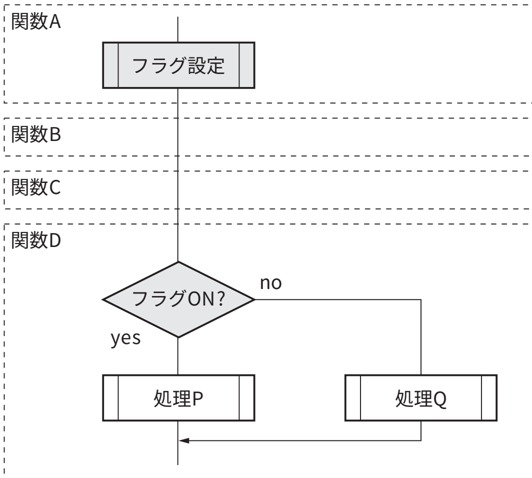


図1　関数間で外部変数フラグを持ち合っていると見通しが悪くなる

ソース・コードの中をのぞいても，そもそもどれが外部変数なのかがわかりにくいことが多い．各関数の挙動が入力パラメータだけで決まらないのでソース・コードが読みにくい．そのため，デバッグがめんどうになる．それだけでなく，テスト・ケースが作りづらく，設計仕様書への反映も困難．

そもそも外部変数というのは，ほかの変数とどこが違うのでしょうか．また，なぜ外部変数は必要なのでしょうか．

● 外部変数の性質を理解する

外部変数は昔から困り者の典型とされています．とはいえ，理由がなければ使われなかったはずです．まず，なぜ必要とされたのかを，外部変数の性質から整理してみることにしましょう．外部変数があふれないようにするには，何に置き換えるべきだったかを整理することから出発します．

外部変数には，以下の二つの特徴があります．

1）ほかの関数から見える

最初に外部変数の特徴として挙げられるのは，関数の引き数にとらなくてもリード/ライトできるというものです．これは，便利な反面，関数間の依存性を高めてしまうので，デバッグがたいへんめんどうになるという欠点を含んでいます．

例えば，ある外部変数をON/OFFの2値のフラグにして，処理を切り分ける関数を考えてみましょう．この関数は，関数単体での動作確認としてはON/OFF時にそれぞれ正しく動作すればOKとすることができます．ところが，機能ブロック全体が正しく動くためには，その手前の関数でON/OFFが正しく設定されていることが絶対条件になります．つまり，単体としては動作確認OKとしたはずの関数が，結合するとほかの関数の挙動しだいでおかしな動きをすることになってしまうのです．

これでは，テスト・ケースも作りづらくなります(**図1**)．

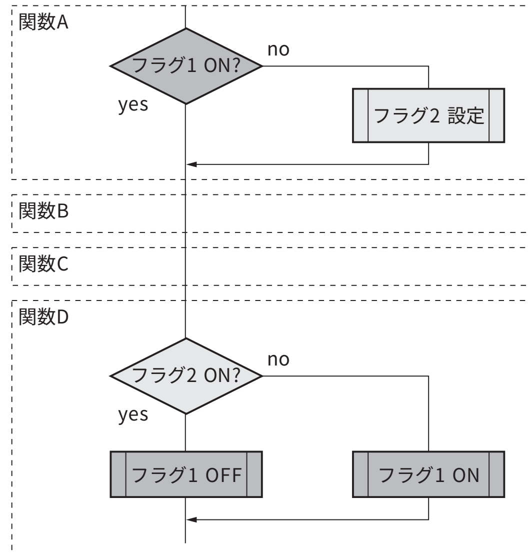


図2　値が保持されて使い回されるとさらにややこしい

関数の動作仕様が1回の呼び出しで完結しないため，デバッグもテスト・ケースの作成も困難になる．どの外部変数がどの関数で更新されて値が書き変わるのかよくわからない場合，ソース・コードを書いた人を含めて，だれにも保守できない化け物を生み出しかねない．

ほかの関数の挙動に左右されてしまう作りであるため，単体テストを行っても関数の動作保証が不十分となってしまうからです．

2）値を保持する

もう一つの特徴は，外部変数は一つの関数が呼ばれて戻ってもその値を保持し続けるというものです(**図2**)．これは，**1)**の特徴とあいまってプログラムの保守性を低下させます．なぜなら，関数による一つの処理が終わった時点で「状態」を記憶する変数があるため，関数を呼ぶたびに動作が変わってくる可能性がある，ということになるからです．

● さらに深く「なぜ」と問うと

ある関数からアクセスする外部変数が一つしかないときは，まだ話はそれほど込み入ってきません．ややこしいのは，外部変数を使いたい放題のプログラムの場合です．このような外部変数のフラグは，たいてい一つではすまなくなっています．

複数のフラグが互いに依存しあって設定されているとき，「正しい」外部変数の設定というのは，機能ブロックの外部仕様から考えるとどのようなものになるのでしょうか．これを明確に定義しようとするとひと言ではすみません．かといって，この手のプログラムの挙動を把握するために「外部変数仕様書」の類を起こそうとすると，小さな機能ブ

ロックでも数十ページのドキュメントになりそうです．

さらに進めて，これらの外部変数はなぜ外部変数として実装されているのかを考えてみましょう．例えば，値を保持するという二つ目の性質が不要であれば，呼び出し元の関数のローカル変数かなにかにフラグをとって，引き数で渡していってもよいはずです．また，必要ないとすれば，どのようなことがあらかじめ明らかになっていればよいのでしょうか．

2 外部変数に対する解決策

では，ここから解決策を考えることにしましょう．外部変数が無秩序に使われているプログラムがあったとして，それをどのように改善すればよいのでしょうか．

● コーディング規約だけでは焼け石に水

すぐに思いつく管理方針としては，

- 外部変数はかならず引き数にとる
- コーディング規約を定める

といったことが挙げられそうです．しかしこれは，たいていの場合，焼け石に水となってしまいます．

実は，外部変数の無秩序な使用からくる混乱が生じていた場合，まちがいなく設計の方針を立てるレベルで外部変数の管理方針が定まっていなかった点に根本の原因があるからです．このような場合，単なるコーディング規約の採用では混乱は回避できません．たとえルールを定めたとしても，外部変数の管理方針が定まっていなければ，ルールを破ってコードを書く者が現れるからです．そして，遵守を徹底してもプログラムが見通しの悪いものになる点は改善されないのです．

● 解決にはデータ構造の把握が不可欠

つまり，開発しようとする機能ブロックまたはシステムではどのような性質のデータを扱っており，それをC言語あるいはそのほかの言語で実装する場合にどの種類の変数を使えばよいのかということが，設計の段階で方針として定まっており，それにのっとってコーディングされていなければならないのです．以下に，解決策の具体例として筆者がよく用いるデータの管理方針を紹介します（**図3**）．プログラム言語はC言語を用いることとします．

外部変数の管理方針を決める前に，まずプログラムで扱われるデータ領域にはどのような種類が存在するのかを考えていくことにします．システム固有の条件も加味したうえで，データにはどのような種別があるのかを考え，適した管理方針を模索していきたいと思います（**表1**）．

表1　各メモリ領域の特徴

大項目	中項目	小項目	解　説
プログラム	テキスト		—
データ	値を保持	スタティック	リード/ライト
		テーブル	リードのみ
	使い捨て	ダイナミック	関数間で共用
		ローカル変数	1関数のみで使用

● まず，値を保持するデータ領域かどうかで分類する

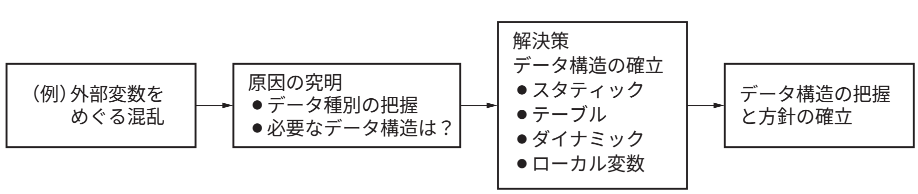
前述したように，外部変数の特徴の一つに「値を保持できる」というものが挙げられます．一方，ローカル変数などは関数を抜けてしまえば値が保持されず，使い捨てとなる領域です．

開発対象のモジュール全体を考えた場合，モジュールのインターフェースとなる関数を抜けても値が保持されなければならないようなデータは，外部変数などの領域をとる必要があります．つまり，データの領域は大きく「使い捨て」と「値を保持」の二つの部分に分けられます（**表1**の中項目）．このどちらに分類されるかで，どのようなデータ種別を用いなければならないかが決まります．一般に，使い捨てのデータ領域のほうが，その重要性と扱いをまちがったときの混乱が大きく，特別な配慮が必要となります．

● 値保持データはスタティックとテーブルに分かれる

次に，値を保持するデータについてさらに詳細を追って

図3
データの管理方針
まず，どのようなデータ構造がそのモジュールにとって必要とされるのかを考える．そして，用いる言語（C言語など）でどのようなデータ種別を使い分けると目的が果たせるのかを調べる．その後，データ構造を設計レベルで確立する．



みることにしましょう．リードのみ（読み取り専用）で，値の更新を予定しないデータがプログラムに用いられることがあります．これは，組み込みソフトウェアでは「テーブル」と呼ばれるデータ領域です．例えば，三角関数の値のように，いちいちプログラムで計算すると処理量がむだなため，あらかじめ計算結果をテーブル引きできるようにしておく場合などに用いられるものです．

このテーブルとは逆に，リード/ライトがともに行われるデータもあります．例えば，ライブラリ・モジュールの最上位関数が呼び出し元のプログラムから呼び出されるたびに値を更新して保持しておかなければならないようなデータです．このようなデータをここでは「スタティック・データ」と呼び，テーブルとは区別しておくことにします（スタティック・データを「RAM領域」，テーブルを「ROM領域」と呼ぶこともある）．

これらがどれだけの大きさを持つかが，組み込みシステム開発では重要になります．管理方針が定まっていなければ，大きさの予想さえ行えません．

● 使い捨てデータはローカル変数とダイナミックの二つ

次に，使い捨てのデータ領域について考えていきましょう．使い捨てのデータは，通常，ローカル変数と呼ばれる変数で実装されます．ほとんどのシステムにおいて，使い捨てデータはローカル変数のみ考えておけばよいと思われています．しかし，組み込みシステムのようにメモリ・リソースが極めて限られている場合は，そうもいかないケースが多々あります．以下に，使い捨てデータ領域についての注意点をまとめます．

1）スタック・オーバフローに注意

ローカル変数はリンカによってスタックと呼ばれる領域にとられます．メモリ・リソースが限られた環境の場合，ローカル変数をむやみにとると，スタック・オーバフローが起きることがあります（**図4**）．

例えば，関数の中に領域長の大きな配列がいくつも使われていた場合，このスタック・オーバフローが起こりえます．ややこしいことに，このスタック・オーバフローは，モジュールの単体テストでは出現しないのに，モジュールを呼び出し元のシステムに結合すると初めて出てくることがあります．スタックは関数の呼び出しが積み重なるたびに積み上げられていきます．そのため，疑似メイン部のすぐ下にモジュールが位置しているので，単体テストでスタック・オーバフローを起こすことはめったにありません．しかし，結合されると関数階層の深いところに位置するようになるため，スタックを使い切ってしまうということが起こりえるのです．

2）OSに頼れない開発環境もある

読者のみなさんの中には，「スタック・オーバフローが起これば，OSが警告を出してアプリケーションが強制終了されるのではないか」と思う方がいるかもしれません．しかし，組み込みソフトウェアの開発環境では，そのような機能を備えたOSを用いているとはかぎりません．そのような場合，目に見える現象としては，単なるシステムのハングアップとしか映らないからやっかいです．スタック・オーバフローがハングアップの原因だと気づくまでには，かなりの時間がかかることでしょう（ハングアップ時の切り分けについては，本特集第4章を参照）．

そのため，ここではローカル変数とは別に，要素数の大きな配列などは「ダイナミック・データ」と呼ばれる領域に集めてしまい，モジュールの中の関数ごとに共用するようにします（**図5**）．ローカル変数は，配列でない変数に限定されることになります．こうすると，スタック・オーバフローが防げるからです．使い捨てのデータ領域にもローカル変数とダイナミック・データの2種類が存在することになります．なお，スタックの消費量は，ローカル変数の総量でのみ決まります．

このように大きな使い捨て領域を集中管理することによって，スタック・オーバフローを防ぐことができます．それだけでなく，関数ごとに領域を使用する「寿命」が重ならないと判明した場合には，同じ領域を使い回すようにする

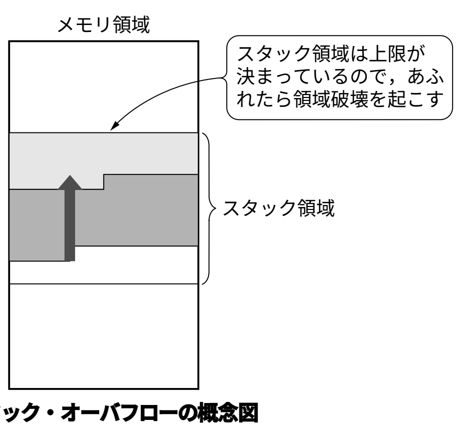


図4　スタック・オーバフローの概念図

OSがあれば警告を発してアプリケーションを強制終了させられる．しかし，組み込みシステムではOSにそのような機能が備わっていないものもある．その場合，すぐには気づかないため，デバッグがたいへんになる．

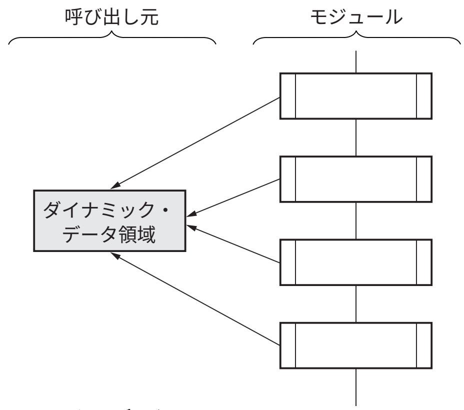


図5　ダイナミック・データ

各関数に大きな配列が散らばっていると，全体でスタック・オーバフローを起こしかねない．対策として，各関数のローカル配列はダイナミック・データ領域に集めてしまう．これらの領域は，処理に用いる寿命が重ならなければ使い回しできるので，領域の節約になる．

ことができるので，メモリ・リソースが限られた組み込みプログラムを作る際には適した設計方針といえます．

3 DSPソフト開発のデータ管理方針

ここからは少しディジタル信号処理プロセッサ（DSP：digital signal processor）のソフトウェア開発に限定して話を進めたいと思います．

ここまで解説したようなデータ構造の一般論が，やや特殊な組み込みシステム（DSPのソフトウェア開発）でどのように扱われるかを解説しながら，データ構造設計の具体例を紹介したいと思います．

この分野では，多くの場合，フィルタ演算と呼ばれる信号処理特有の演算処理の実装が開発のかなめとなります．このような演算処理は，プログラムのほかの部分よりも処理量を費やすものであるため，最適化設計の際にはこの部分の削減が重要となるからです[注2]．

このフィルタ演算をブロック図で描き表したのが**図6**です．これは，FIRフィルタと呼ばれる演算処理を図示したものです．

● FIRフィルタのデータ構造を考える

DSPでは，過去の信号を一定数ため込み，定められた数

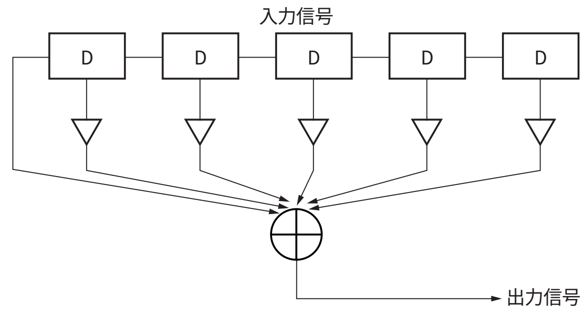


図6　DSPフィルタ演算のブロック図（FIRフィルタ）

FIRフィルタのような計算方法は，積和演算と呼ばれる．信号処理アルゴリズムはそのほとんどがこの積和演算を実行するものである．DSPはそのための専用CPU．

をそれぞれ掛け算して最後に全部足し合わせるという「積和演算」と呼ばれる演算処理を多数行います．歴史的には，むしろこの積和演算に適したアーキテクチャ（ハーバード・アーキテクチャ）を採用した演算処理装置として開発が進んだものが今日のDSPになります．FIRフィルタの演算は積和演算のもっともシンプルなものです．

図6で「D」と書かれているブロックは過去の信号を表しています．**図6**では過去五つ分の信号が保持されていることになります．これらに一定の値を掛け（図の三角の部分），最後にそれらをすべて加算します．実際にプログラムを作る際には，スタティック・データで図の「D」の部分を，テーブルで図の三角形の部分を実装することになります．それ以外の計算のための使い捨て領域はスタック，あるいはダイナミック・データに置かれることになります．

プログラムのデータ構造を決める場合，実装対象のシステムがどのようなデータ構造を必要とするのかを把握し，それらに適した実装方針を定めることが重要です．もちろん，データ構造は単にプログラムの見やすさを高めるために定めるのではありません．例えば，プログラムの処理量を削減する最適化においても，データ構造は大きな影響を与えます．外部変数があふれたからコーディング規約を定めた，というような表層的な対応では，品質の高いシステムは作れないのです．

● 各データ領域における管理方針を確立する

次に，DSPのソフトウェア開発で定めるデータ構造を例にとり，データの管理方針を示します．

表2に各領域の管理方針の例を挙げました[注3]．スタティック・データ領域とダイナミック・データ領域は頻繁に

注2：本稿の続編は，本誌の次号以降に連載記事として掲載する予定．最適化設計については，連載の中で詳細を説明するので，そちらを参照していただきたい．
注3：ここで紹介したデータ構造の決めかたは，DSP向けのソフトウェア開発で比較的広く知られた方法を筆者なりにアレンジしたものである．

表2 各メモリ領域の意味と管理方針を押さえる

小項目	領　域	領域確保	データの持ちかた	注意事項
スタティック・データ	BSS(block started by symbol)/ヒープ	呼び出し元	構造体でたどれる	初期化が必要
テーブル	定数	ライブラリ内	外部変数	大文字にする
ダイナミック・データ	PRAGMA命令などで専用領域確保	呼び出し元	構造体でたどれる	ローカル配列を集める 隠れスタティックに注意

リード/ライトを行う場所なので，性能を向上させるためには内部RAMのようにアクセス速度の速いメモリ領域に確保できなければなりません．そのため，それぞれの領域長がどれだけ必要かも重要となってきます．ですので，ライブラリ・モジュールの中に置くよりはその呼び出し元で領域を確保し，関数呼び出し時に使用アドレスをポインタで渡すようにするとよいでしょう．ライブラリ・モジュールは，渡されたポインタを構造体でキャストするようにします．つまり，スタティック・データ用とダイナミック・データ用の構造体を先頭にしてアドレスをたどれるようにすれば，呼び出し元は配列の形で必要な領域を確保するだけでよくなるということです（**図7**）．この配列は，グローバル領域かヒープ領域に配置します．

●「隠れスタティック・データ」に注意

スタティック・データを扱う場合，注意しなければならないのは，関数呼び出しにまたがって値を保持するものであるため，かならず初期値が定められていなければならないということです．つまり，スタティック・データのあるモジュールには，かならず初期化関数を用意しておく必要があるのです．もし初期化されていないと，不定値で処理を開始することになるため，プログラムの挙動がおかしくなります．

ダイナミック・データで気をつけなければならないのは，誤ってスタティック・データをダイナミック・データとして扱ってしまうことです．関数の呼び出しにまたがって値を保持していなければならないのに使い捨て領域に配置してしまうと，当然プログラムは正しく動作しません．ところが，単体テストでこれに気づかないこともありえます．見逃すと，「結合時に動かない」というはめになるのです．

以上の理由から，スタティック・データとダイナミック・データには特別のテストが必要となります（**表3**）．具体的には，それぞれの領域を0xAAAAとか0x5555といった値で埋めるテストを行います（**図8**）．スタティック・データは初期化の手前で，ダイナミック・データは通常処理の関数の手前で毎回初期化するようにします．こうすると，

表3　各メモリ領域のテスト方針

小項目	テスト方針
スタティック・データ	初期化の手前で領域を埋めて動作
ダイナミック・データ	関数呼び出しの手前で毎回領域を埋めて動作

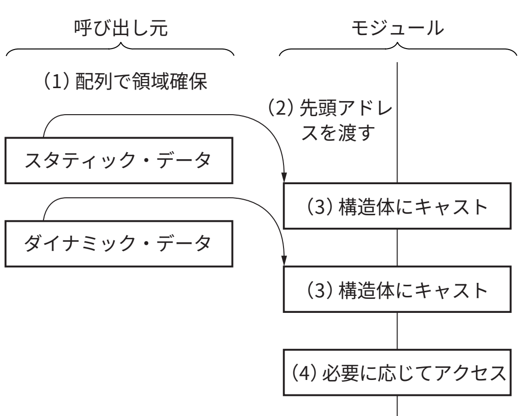


図7　データの管理

呼び出し元の関数は，スタティック・データ用またはダイナミック・データ用の領域を配列で確保する．配列はグローバル領域かヒープ領域にとっておき，先頭アドレスを渡す．呼び出された側は，先頭アドレスに構造体をキャストすれば必要なデータに自在にアクセスできる．

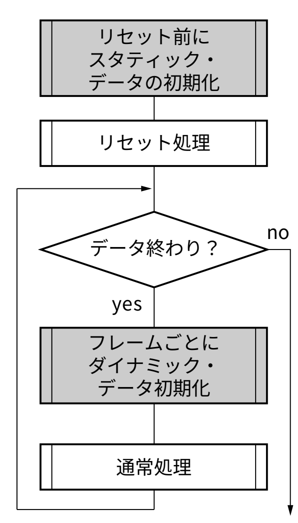


図8 ダイナミック・データとスタティック・データの確認テスト

スタティック・データを初期化しそこねて，不定値のまま処理していないかを確認するのが目的．ダイナミック・データについては，隠れスタティック・データがないかを確認することが目的．初期化（図の灰色の部分）で出力がおかしくならなければOK．

5

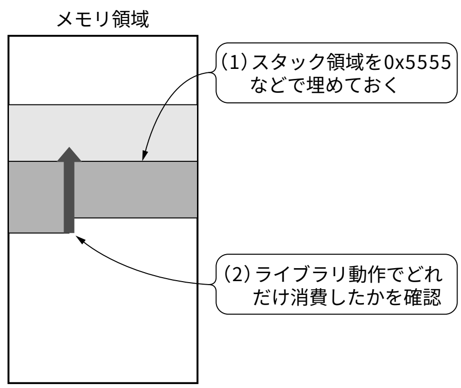


図9　スタック・オーバフローの確認テスト

DSPのソフトウェア開発では，リンク時のメモリ・マッピング情報を調べることができる．そのため，スタックに用いる領域はあらかじめわかるので，それらをすべてなんらかの値で埋めつくして余りが出るかを調べる．

初期化漏れで不定値を初期値にしてしまうミスや，スタティック・データを誤ってダイナミック・データ扱いしてしまうというミスが見つかります．

テーブルの場合，スタティック・データやダイナミック・データとは異なり，外部変数としてライブラリ内に配置します（呼び出し元に領域を持たないということ）．テーブルは，読み出し専用の領域なので，外部変数として確保しても可読性や保守性が落ちてしまうことはないからです．ただし，そのデータがテーブルであることがわかるように，「テーブルはかならず大文字にする」などのコーディング規約を設けておくとよいでしょう．

● スタック・オーバフローの予防テスト

前述のように，ローカル変数に大きな配列をとってはなりません．そうすることで，スタック・オーバフローを招く事態が防げます．

もっとも，実際にスタック・オーバフローを起こしていないかを確認するテスト・ケースは別途必要となります．具体的には，スタック領域を0xAAAAとか0x5555といった値で埋めてテストを実施することになります（**図9**）．DSPのソフトウェア開発では，リンク時のメモリ・マッピング情報を調べることができるので，こうしたテストが可能です．スタックは，通常のCPUではアドレスをさかのぼって消費していくのですが，それがどこまでたどり着いたかを，プログラムを動作させた後に確認すれば，スタック使用量がどれだけあるかを実測できるからです．

4 アカデミックな知識について

ここで，ソフトウェア工学でいうところの本来（？）のデータ構造の議論についてひと言．大きな書店の情報科学のコーナを訪れると，「データ構造うんぬん」という題名の書物がいろいろと並んでいます．筆者はコンピュータ・アルゴリズムの専門家ではありませんから，そのすべてに精通しているわけではありませんが，実装において不可欠と思われるデータ構造の検討方法が書かれている本はあまりないように思います（そもそも，アルゴリズムの専門家とは関心分野が違うのだから，筋違いな要求だと言われればそれまでだが…）．

● 力点はシステムによって異なる

本稿で紹介したようなデータ構造の管理方針を読んで，「ずいぶん変わったやりかただな」と感じられた方もいるかもしれません．ここで紹介した管理方針はあくまでも1例なので，読者のみなさんが開発を行う際には自分のシステムに合ったデータ構造を考えるようにしてください．自分のシステム設計のデータ構造を決める場合，まず実現しようとするアルゴリズムや機能などがどのような性質のデータを必要としているのかを十分に考察してから始めていただきたいと思います．

例えば，組み込みシステムのプログラムであっても，メモリ量を十分にとれるシステムもあります．そのような場合，ローカル変数によるスタック・オーバフローを心配する必要はありません．

また，スタティック・データの領域確保は，本稿のように呼び出し元で最初に確保しないという設計もありえます．例えば，ライブラリの中の初期化処理において，最初に領域を必要なオブジェクトの数だけ確保しておき，終了処理で削除する方法などが考えられます．

要するに，システムによってデータ構造設計時の力点の置きかたやテスト方法は異なります．みなさんのシステムに適したデータ構造を考える際には，ここで紹介したような管理方針の趣旨をよく理解したうえで，自分のシステムに合ったデータ構造を確立してみてください．

さえき・はじめ